# *Eventual Consistency*

## Readings

- Werner Vogels ACM Queue paper
  - http://queue.acm.org/detail.cfm?id=1466448
- Dynamo paper
  - http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- Apache Cassandra Consistency docs
  - http://www.datastax.com/documentation/cassandra/2.1/cassandra/dml/dmlAboutDataConsistency.html

# *Eventual Consistency*

- Recall: data is replicated for performance and failover purposes
  - If lots of clients want to read same data, can read off diferent replicas
  - When a machine goes down, the data is still available somewhere else

# *How to implement R/W operations*

- Ideally, want accesses to a single object to be atomic/**linearizable**
- I.e. if several processes are using the same object, it should "look like" they were accessing it in sequence on a single-node system

## *Linearizability*

- Linearizability requirements
  - A total order on operations across the whole system
  - Consistent with wall-clock ordering for non-overlapping operations
  - If a read is ordered after a write, the read should see the value written by that write (or a later one)
- Sometimes called consistency, but not the same kind of consistency as ACID consistency

## The Problem

- Want to enforce this without sacrificing availability
  - Users must be able to perform reads/writes when they want to
- Plus, things get even worse if your system gets partitioned due to network failures
  - If you really want to keep replicas in sync, must block everyone until network is back up…

## *Consistency vs. Availability*

- A fundamental tradeoff between
  - Consistency
  - Availability
  - Partition Tolerance
- "Brewer's Conjecture" or "CAP theorem"

## *Consistency*

- (As seen before): accesses to a single object must be atomic/linearizable

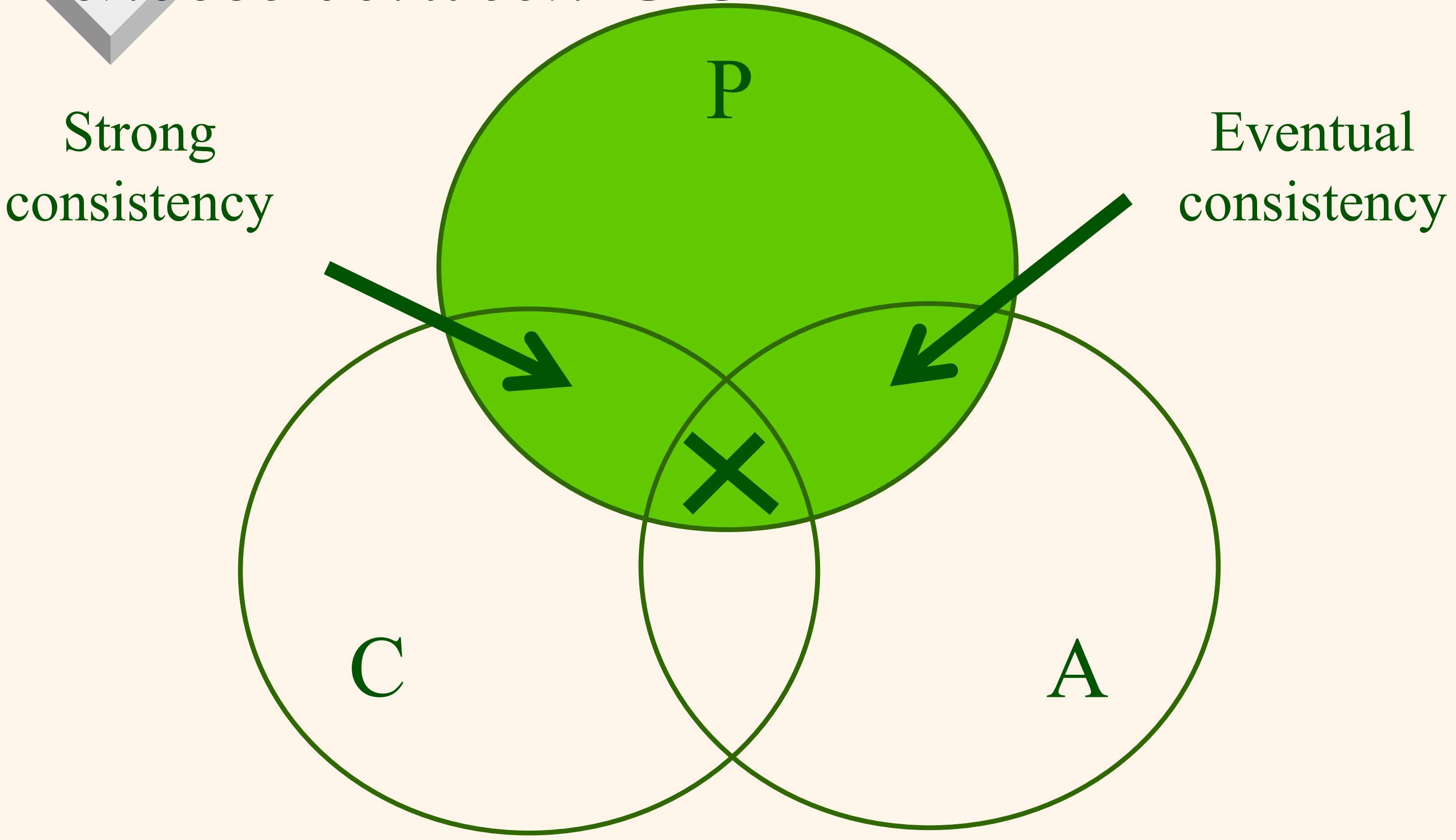
## *Availability*

- All clients must be able to access some version of the data
  - Every request must (eventually) receive a response
- Example: every write must succeed, cannot be forgotten or rejected by system

## *Partition Tolerance*

- The system must tolerate network partitioning scenarios and still function correctly
- Realistic and sensible requirement; network partitions occur all the time in the real world once your system is large enough

## *If partitions are unavoidable, must choose between C & A*



## *The tradeoff*

- Some modern systems choose A over C
  - Many of them NoSQL
  - Makes sense – originally built for use cases where scalability and availability are paramount, and strong consistency and other features traditionally provided by an RDBMS are less crucial

# *Eventual Consistency*

- System must always be able to take reads and writes
- Even when network becomes partitioned
- Consequence: can no longer guarantee all replicas kept in sync
  - And programmer must work with that!
- Replicas do (eventually) get back into sync, but there is a definite window of inconsistency

## *EC – the System Perspective*

- Three important parameters:
- N: Replication factor
  - How many copies of the data are stored?
- R: How many replicas are checked during a read
- W: How many replicas must acknowledge receipt of a write

## *Relationship between N, R and W*

- If W + R > N, can guarantee strong consistency
  - Read and write set always overlap
- Various tradeoffs can make sense
  - W = N, R = 1 achieves fastest reads
  - W = 1, R = N achieves fastest writes
  - Can also do W=Q, R=Q where Q = N / 2 + 1 ("Quorum")

## *EC in Apache Cassandra*

- NoSQL system
- Open source
  - Initially developed by Facebook
- Column family data model
- Provides tunable consistency guarantees
  - So you can trade off consistency for performance (availability) as you want by setting the appropriate `ConsistencyLevel`

## Cassandra Writes

| Level | Behavior |
|---|---|
| ANY | Ensure that the write has been written to at least 1 node, including HintedHandoff recipients. |
| ONE | Ensure that the write has been written to at least 1 replica's commit log and memory table before responding to the client. |
| TWO | Ensure that the write has been written to at least 2 replicas before responding to the client. |
| THREE | Ensure that the write has been written to at least 3 replicas before responding to the client. |
| QUORUM | Ensure that the write has been written to N / 2 + 1 replicas before responding to the client. |
| LOCAL_QUORUM | Ensure that the write has been written to <ReplicationFactor> / 2 + 1 nodes, within the local datacenter (requires NetworkTopologyStrategy) |
| EACH_QUORUM | Ensure that the write has been written to <ReplicationFactor> / 2 + 1 nodes in each datacenter (requires NetworkTopologyStrategy) |
| ALL | Ensure that the write is written to all N replicas before responding to the client. Any unresponsive replicas will fail the operation. |

## Cassandra Reads

| Level | Behavior |
|---|---|
| ONE | Will return the record returned by the first replica to respond. A consistency check is always done in a background thread to fix any consistency issues when ConsistencyLevel.ONE is used. This means subsequent calls will have correct data even if the initial read gets an older value. (This is called ReadRepair) |
| TWO | Will query 2 replicas and return the record with the most recent timestamp. Again, the remaining replicas will be checked in the background. |
| THREE | Will query 3 replicas and return the record with the most recent timestamp. |
| QUORUM | Will query all replicas and return the record with the most recent timestamp once it has at least a majority of replicas (N / 2 + 1) reported. Again, the remaining replicas will be checked in the background. |
| LOCAL_QUORUM | Returns the record with the most recent timestamp once a majority of replicas within the local datacenter have replied. |
| EACH_QUORUM | Returns the record with the most recent timestamp once a majority of replicas within each datacenter have replied. |
| ALL | Will query all replicas and return the record with the most recent timestamp once all replicas have replied. Any unresponsive replicas will fail the operation. |

# Amazon Dynamo

- A key-value store created by Amazon in the mid 2000-s
  - Academic paper in SOSP 2007
- One of the big names in early NoSQL and EC systems
- Designed from the ground up to serve Amazon's unique needs
- Notably: system must always be able to take a write
  - E.g adding item to shopping cart
  - Otherwise sales are lost

## *Dyamo conflict resolution*

- Dynamo is an EC system
- A put() call may return to its caller before the update has been applied at all the replicas
- A get() call may return many versions of the same object.
- These versions must be reconciled at some point
  - Your shopping cart had better be consistent at checkout!!

# *Dyamo conflict resolution*

- When to reconcile?
  - At write time? -> not good, slows down writes
  - So, need to reconcile versions at read time
- Who should reconcile?
  - Datastore? -> doesn't know enough about your application semantics
    - Can only use simple policies like "last write wins"
  - So, conflict resolution is up to the application

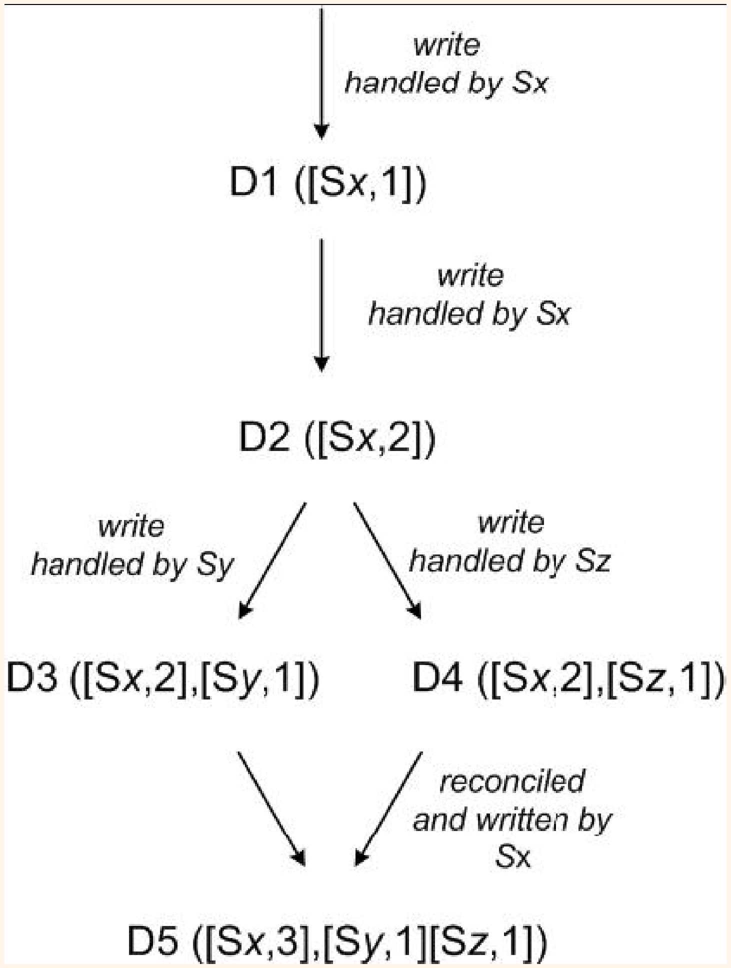
## *Data Versioning*

- Challenge: an object having distinct version sub-histories, which the system will need to reconcile in the future.
- Solution: uses vector clocks in order to capture causality between different versions of the same object.

## *Vector Clock*

- A vector clock is a list of (node, counter) pairs.
- Every version of every object is associated with one vector clock.

# *Vector Clock Example*



## *Vector Clock*

- Every version of every object is associated with one vector clock.
- If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten.

## *EC – the client perspective*

- What guarantees can your code expect?

- Strong consistency – after an update completes, any subsequent access returns updated value
- Weak consistency – above is not guaranteed.
  - Window of inconsistency during which older values may be seen

## *Kinds of consistency*

- Eventual consistency – a form of weak consistency
  - the system guarantees that – if no new updates are made – eventually all accesses will return latest value
- When is "eventually"?
  - Depends on a lot of system parameters:
    - Number of replicas
    - Load on system
    - Communication delays…

## *Kinds of eventual consistency*

- Read-your-writes consistency
  - If a process has written a data item and reads it later, should see own update
- Session consistency
  - Read-your-writes consistency in the (restricted) context of a session

## *Kinds of eventual consistency*

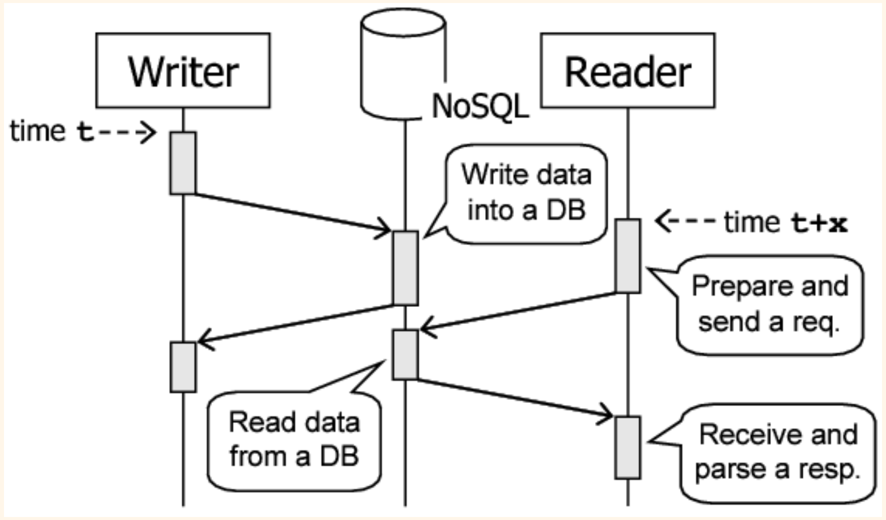
- Monotonic write consistency: writes by a single process are serialized in order
- Monotonic read consistency: if a process has seen a value, will never see an older value on subsequent reads
- These can be combined in various ways

## *How well does this work it in practice?*

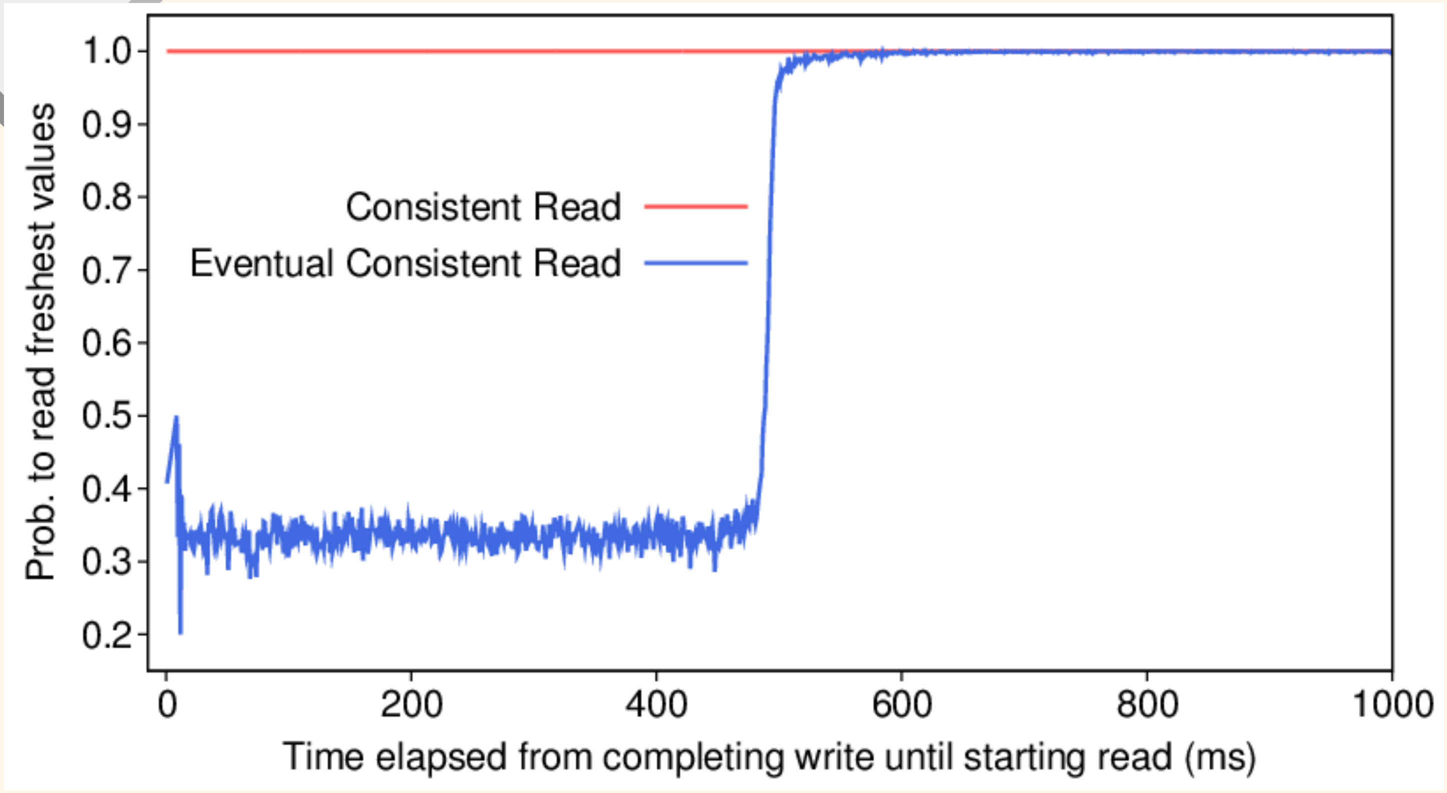
- Depends on the specific system
- Experimental investigations
  - *Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers' Perspective*, H. Wada et al, CIDR 2011
  - Experiments on Amazon SimpleDB
    - Supports consistent reads and eventually consistent reads
    - "Consistency across all copies of the data is usually reached within a second"

# *Frequency of Observing Stale Data*

- Experimental Setup
  - A writer updates data once each 3 secs, for 5 mins
  - A reader reads it 100 times/sec
    - Check if the data is stale by comparing value seen to the most recent value written
    - Plot against time since most recent write occurred
  - Experiments carried out in Oct/Nov, 2010

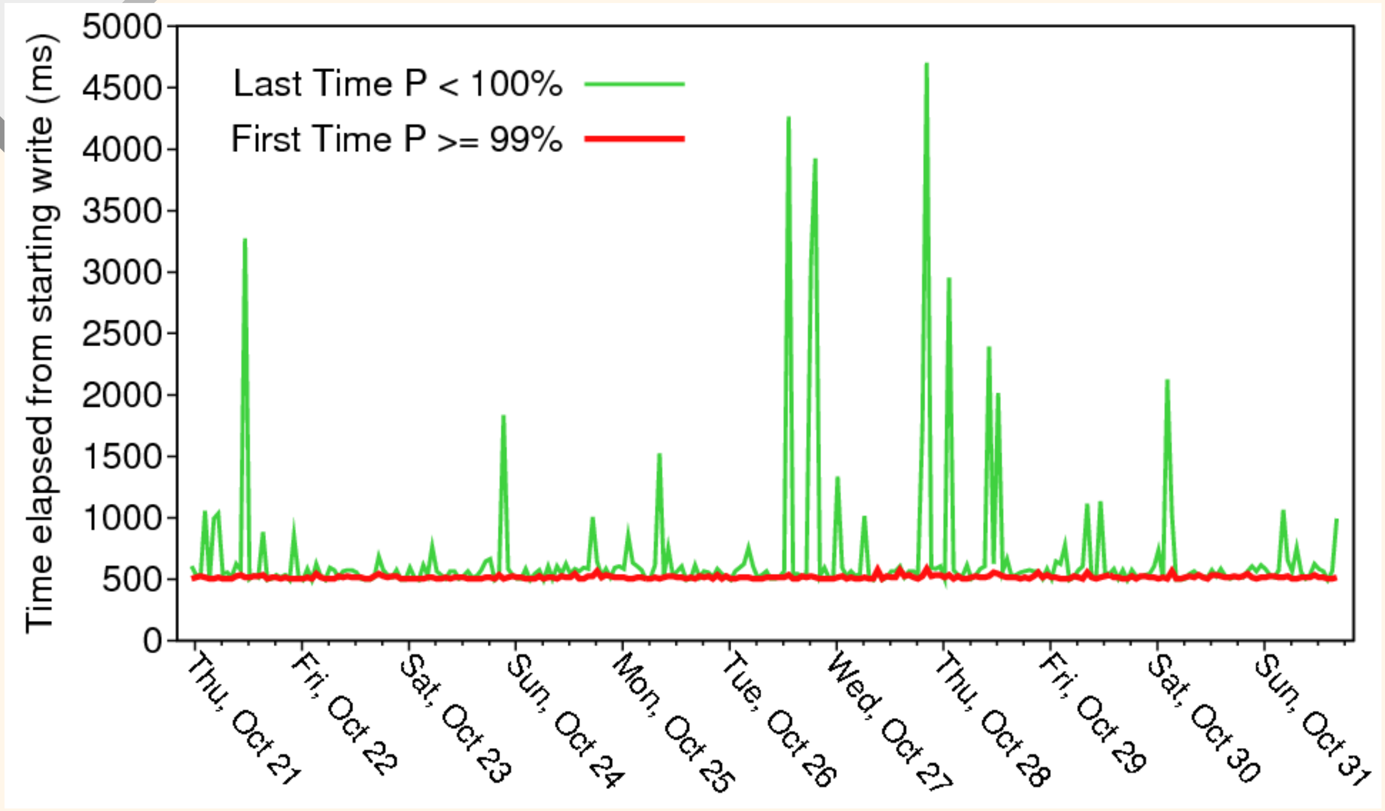


# *Read and Write from a Single Thread*



- With eventually consistent read, 33% of chance to read freshest values within 500ms
  - Perhaps one master and two other replicas. Read takes value randomly from one of these?

## *Read and Write from a Single Thread*



- First time for eventual consistent read to reach 99% "fresh" is stable 500ms
- Outlier cases of stale read after 500ms, but no regular daily or weekly variation observed

# *Monotonic Reads*

- Definition: each read sees a value at least as fresh as that seen in any previous read from the same thread/session
- Experiment: check for a fresh read followed by a stale one (within a period of <450 ms from write)
- SimpleDB ec reads do not provide this guarantee!
  - In staleness, two successive eventual consistent reads are almost independent

| | 2nd Stale | 2nd Fresh |
|---|---|---|
| 1st Stale | 39.94% | 21.08% |
| 1st Fresh | 23.36% | 15.63% |

## *Eventual Consistency Summary*

- A tradeoff to privilege availability (ability to execute reads and writes) over consistency
- Not appropriate in all cases and for all applications
  - Need to decide if temporary inconsistency is acceptable
  - Generally much harder to program on an ec system
- Unexpected effects can and do occur
  - https://aphyr.com/posts/322-call-me-maybe-mongodb-stale-reads for a recent article on MongoDB issues